Hoare-Style Reasoning with (Algebraic) Continuations

Germán Andrés Delbianco
IMDEA Software Institute
Universidad Politécnica de Madrid
german.delbianco@imdea.org

Aleksandar Nanevski
IMDEA Software Institute
aleks.nanevski@imdea.org

Abstract

Continuations are programming abstractions that allow for manipulating the "future" of a computation. Amongst their many applications, they enable implementing unstructured program flow through higher-order control operators such as callcc. In this paper we develop a Hoare-style logic for the verification of programs with higher-order control, in the presence of dynamic state. This is done by designing a dependent type theory with first class callcc and abort operators, where pre- and postconditions of programs are tracked through types. Our operators are algebraic in the sense of Plotkin and Power, and Jaskelioff, to reduce the annotation burden and enable verification by symbolic evaluation. We illustrate working with the logic by verifying a number of characteristic examples.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Control structures; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Logic of programs, Pre- and post-conditions; F.3.3 [*Studies of Program Constructs*]: [Control Primitives]

General Terms Languages, Verification

Keywords Continuations, Hoare Logic, Dependent Types, callcc

1. Introduction

Continuations are powerful abstractions that model the "future of a computation" [36]. They have a ubiquitous presence in programming languages: they allow for a family of program transformation techniques in the style of many CPS transformations [11], they underlie the denotational semantics of programs with jumps [16, 42], they give computational content to classical proofs [20], they have been used to structure computational effects [17, 21] and also to design compilation techniques. Moreover, certain programming languages provide first-class control operators which manipulate continuations, e.g. the variants of **callcc** in Scheme, ML and Haskell, or the related $\mathcal{C}$ and $\mathcal{F}$ control operators [15, 14].

The ability to manipulate "the future" makes these operators more powerful than plain **goto**-like instructions, but it also hinders the formal reasoning about programs. Although the state of the art concerning formal reasoning about continuations is vast, it has focused predominantly (with notable exceptions discussed below) on the semantic modelling of higher-order control operators and CPS transformations [15, 20, 11, 47], and verification of programs directly in the semantic model [41, 13].

In contrast, in this paper we are interested in developing a Hoare-style logic in which one can systematically specify and verify full functional correctness of programs with higher-order jumps, in the presence of dynamic mutable state and the ability to capture continuations and return them as results of computations, potentially encapsulated into closures.

The ability to capture and return continuations makes our task more difficult when compared to the previous work on Hoare logics for first-order jumps (i.e. **goto**'s) in high-level languages [9, 25, 2] and low-level machine code [38, 46, 23]. In particular, the higher order nature of **callcc** entails the need for a Hoare logic capable of reasoning about (potentially higher-order) functions. It also makes it somewhat more difficult to design the specification methodology, i.e. decide on just what kind of information should the proof developer provide in the form of annotations when specifying a program involving **callcc**, and how should that information relate the context in which the continuation is captured to the context in which it is invoked.

The presence of dynamic state significantly complicates matters, and differentiates our work from recent Hoare-style logics for higher-order jumps [5, 10]. From the semantic point of view, supporting dynamic state requires building a model in which executing a continuation does not roll back the mutable state to the point at which the continuation is captured. From the specification point of view, it requires reconciling **callcc** with Separation logic [31, 37]. In this paper, we accomplish the task in a novel manner, combining Separation *assertion* logic with large footprint semantics and large footprint inference rules for verification in the style of symbolic evaluation.

More concretely, our contribution in this paper is the development of $\mathsf{HTT}_{\mathbf{cc}}$, a framework for Hoare-style reasoning with higher-order programs, control effects and mutable, dynamic state. To the best of our knowledge, this is the first formal system for reasoning about such combination of features. We define a dependent type-theory with first class continuations which uses monadic types indexed by pre- and postconditions as our specifications in the style of Separation logic. This has been done before for a language with state, e.g. in *Hoare Type Theory* (HTT) [29, 30], but now we extend the approach to a language with continuations. In particular, we rely on *dependent records* (i.e. iterated Σ-types) as an essential tool for specification of programs which "capture the continuation" in a closure that is later invoked.

In order to make specification and verification in $\mathsf{HTT}_{\mathbf{cc}}$ more palatable, we focus on a specific choice of control operators which are *algebraic* in the sense of Plotkin and Power [33, 34] and Jaskelioff [22]; that is, the control operators commute with sequential composition. We argue that the algebraic control operators are less burdensome for verification than the non-algebraic alternatives, be-

ICFP '13, September 25–27, 2013, Boston, MA, USA.

ACM 978-1-4503-2326-0/13/09. . . $15.00.
http://dx.doi.org/10.1145/2500365.2500593

1. $\{x \mapsto v\}$
2. $c \leftarrow \mathbf{callcc}\, f.$
   $\mathbf{ret}\ (\mathbf{abort}\ f\ (x := !x + 1;\ \mathbf{ret}\ (\mathbf{ret}\ ())));$
3. $\{x \mapsto v \curlywedge \{x \mapsto v + 1\}\, \mathbf{c}\, \{\bot\} \curlyvee$
   $x \mapsto v + 2 \curlywedge \{x \mapsto v + 3\}\, \mathbf{c}\, \{x \mapsto v + 3\}\}$
4. $x := !x + 1;$
5. $\{x \mapsto v + 1 \curlywedge \{x \mapsto v + 1\}\, \mathbf{c}\, \{\bot\} \curlyvee$
   $x \mapsto v + 3 \curlywedge \{x \mapsto v + 3\}\, \mathbf{c}\, \{x \mapsto v + 3\}\}$
6. c
7. $\{x \mapsto v + 3\}$

Figure 1. An idealised proof outline for **inc3**.

cause in practice they require less annotations to be manually provided by the user. In particular, the algebraic **callcc** typically requires manual description of only the jumping behaviour of the code, whereas, in our experience, the non-algebraic variants require manual description of both the jumping and the normal termination of the **callcc** block.

To test our design in practice, we have implemented $\mathsf{HTT}_{\mathbf{cc}}$ as a shallow embedding in the Calculus of Inductive Constructions (CIC), as realised in Coq [27, 6] and Ssreflect [19]. We have mechanised its denotational semantics and soundness proofs and used the framework to verify a survey of standard examples which cover the usual continuation idioms. A subset of the examples is presented in Section 6. All of our Coq code is available on-line [12].

2. Overview

The language of $\mathsf{HTT}_{\mathbf{cc}}$ uses monads, as in Haskell, to separate the purely functional and the imperative fragment of the language. In addition to capturing continuations via **callcc** and jumping to them via **abort**, the imperative fragment supports recursion (which we consider a side effect), and heap-mutating commands such as allocation, deallocation, reading from and writing into heap locations. We use $\mathbf{ret}\, v$ for a monadic command returning a value v, and $x \leftarrow e_1; e_2$ for monadic bind (i.e. a sequential composition), which first executes e_1, then binds the return result to x before executing e_2. We abbreviate with $e_1; e_2$ when $x \notin \mathsf{FV}(e_2)$. As customary with monadic languages, we assume that a command is evaluated only upon being bound in a sequential composition. Until then, the command's execution is suspended.

2.1 Algebraicity

As our **callcc** and **abort** are non-standard, we briefly illustrate them by example. Figure 1 shows an *informal* proof outline—in a style of partial correctness Hoare logic or Separation logic—for a function **inc3**, which uses a backward jump to increment the value of pointer x by 3. Our actual syntax for **inc3**, and the manual assertions needed for its specification, will be introduced in Section 2.3. We first discuss the behaviour of the function, and then return to the proof outline.

In the first command (line 2), **inc3** captures the continuation, which, at that point, is $\lambda c.\ x := !x + 1;\ c$, corresponding to the code in lines $4 - 6$, with the variable c abstracted. The continuation is encapsulated inside a *continuation object* f, which may be viewed as a label for jumps. Next, the body of **callcc** is executed, and the suspended command $\mathbf{abort}\ f\ (x := !x + 1;\ \mathbf{ret}\ (\mathbf{ret}\ ()))$ is bound to c.[1] The program continues by incrementing x (line 4), after which c follows (line 6). Execution of c causes a jump to the continuation encapsulated inside f. However, before the control is passed to the continuation, the second argument of **abort**—$x := !x+1;\ \mathbf{ret}\ (\mathbf{ret}\ ())$— is executed. Thus, x is incremented again, and $\mathbf{ret}\ ()$ is passed as the argument c to the continuation encapsulated inside f. Passing the control to the continuation corresponds to a backward jump to line 4. Thus, x is incremented once again, followed by execution of c. Since the latter variable is now bound to $\mathbf{ret}\ ()$, its execution falls through and **inc3** returns $()$, after having incremented x three times.

The non-standard aspect of our control operators is that **abort** allows executing arbitrary side-effectful command—in the above case $x := !x + 1$—as part of performing the jump. This is different from the customary **callcc** and **throw** [28, 48, 49], as the latter only passes a *value* upon a jump. We refer to this side-effectful command as *finalisation* code, as it is executed immediately before the jump, thus ending the normal control flow. Obviously, **throw** can be mimicked by **abort** by using a trivial finalisation code which immediately returns. Dually, $\mathbf{abort}\ f\ e$ can be implemented by sequential composition which executes e, followed by **throw**ing the obtained value to f.

However, choosing **abort** as a primitive, awards special status to the finalisation code, which makes the control operators *algebraic* [22, 33, 34]. More concretely, **callcc** commutes with sequential composition, a property we use in Section 4 to formulate a methodology for Hoare-style specification and verification by symbolic evaluation. We illustrate how the commutation arises by the following equations, which can be derived from Jaskelioff [22].

$$x \leftarrow (\mathbf{callcc}\, f.\, e_1); e_2 = \mathbf{callcc}\, g.x \leftarrow [(g \triangleright x.\, e_2)/f]e_1;\ e_2 \quad (1)$$

$$\mathbf{abort}\ (g \triangleright x.\, e_2)\ e_1 = \mathbf{abort}\ g\ (x \leftarrow e_1;\ e_2) \quad (2)$$

$$\textit{where} \quad g \triangleright x.\, e_2 \triangleq \lambda t.\, g\ (x \leftarrow t; e_2)$$

Consider equation (1). The continuation captured inside f on the left side of the equation includes e_2. On the right side, **callcc** has been commuted out of the scope of x, and thus e_2 cannot be part of the continuation captured inside g. To induce that the expressions on the two sides of equality behave the same, jumps to g on the right side must be *preceded* by an execution of e_2. Because our primitives provide for finalisation code, we could enforce such a discipline by using e_2 as finalisation code for every **abort** to g. This is achieved by uniformly substituting f in e_1 with a new continuation object $g \triangleright x.e_2$. The new object is engineered so that aborting to it object behaves like aborting to g with the finalisation code extended by e_2, as captured in equation (2). Thus, execution of e_2 precedes the jumps to g, just as required.

2.2 Proof outline

As customary in Separation Logic, heaps are finite maps from the type **ptr** of pointers (isomorphic to $\mathbb{N}$) to values. The predicate $x \mapsto v$ holds only of a singleton heap with a pointer x, storing the value v. We use $\curlywedge$, $\curlyvee$, for point-wise conjunction and disjunction of heap *predicates*, and $\top$ and $\bot$ for the always true and always false predicate, respectively. In subsequent examples, we will also use separating conjunction $P * Q$, which holds of a heap h if h can be split into disjoint parts satisfying predicates P and Q, respectively. We will also use the predicate $\mathsf{this}\, h$, which holds only of heaps equal to h. We write $P\ h$ or alternatively, $h \in P$, when the predicate P holds of the heap h. We retain $\wedge$, $\vee$, True and False for the customary propositional (i.e., non-separation) connectives.

Referring to Figure 1, the line 1 states that the program starts with the initial heap containing a pointer x storing the integer v (though we omit the type annotations). $\mathsf{HTT}_{\mathbf{cc}}$ uses *large footprint*

[1] As usual with monads, the binding strips the outer **ret**.

annotations which describe the full heap in which the program runs, rather than just a subheap that the program needs (in contrast to the *small footprint* annotations from Separation logic). Thereby, the proof outline in Figure 1 describes the behaviour of **inc3** when the heap contains *exactly* the pointer x storing v, but no other pointers. For now, we restrict ourselves to this simple case in order to focus on algebraicity, but we explain in Section 3 how to generalise the annotations of **inc3** to cover larger heaps.

Going back to Figure 1, at line 3, after the continuation is captured in line 2, the current heap is unchanged, but the program variable c is bound to the command $\mathbf{abort}\ f\,(\mathbf{ret}\,(\mathbf{ret}\ ()))$, which itself has to be specified. The Hoare triple $\{x \mapsto v+1\}\ c\ \{\bot\}$ indicates that c should be executed only after x is incremented (precondition $x \mapsto v+1$), and that the execution of c causes a jump (postcondition $\bot$). This behaviour precisely corresponds to the intended use for c in line 6. However, as c in line 6 executes a backward jump, the assertion in line 3 has to describe the state right after the jump, but before the program proceeds with executing line 4 for the second time. This is the role of the second disjunct in line 2. It shows that the program point is reached for the second time with x incremented twice. As described in Section 2.1, at that point c is bound to $\mathbf{ret}\ ()$, and can be specified by $\{x \mapsto v+3\}\ c\ \{x \mapsto v+3\}$ to indicate that this second instance of c will be executed after x is incremented once more (precondition $x \mapsto v+3$). The second execution of c does not jump, but falls through with the heap unchanged (postcondition $x \mapsto v+3$).

After x is incremented in line 4, the assertion in line 5 accounts for the change in the heap: compared to line 3, the value of x is incremented in both disjuncts, while the specifications for c remain unchanged. Now, command c is safe to execute in line 6, because the heap in both disjuncts satisfies the respective preconditions for c. The program terminates satisfying $x \mapsto v+3$ (line 7), which is a disjunction of the postcondition of c from line 5.

2.3 Proof annotations as dependent types

The crucial point in the proof outline for **inc3** is deciding on the specifications for c in line 3, which indicate the intended use of c in the rest of the program (i.e., c is executed when x stores $v+1$ and $v+3$, jumping in the first case, and falling through in the second). Such specifications depend on the structure of the rest of the program, and cannot be gleaned solely from the body of **callcc** in line 2. In this paper, we adopt the approach that such information is provided by the programmer in the form of annotations. This is similar to the way loop invariants often have to be provided when verifying structured programs. In $\mathsf{HTT}_{\mathbf{cc}}$, we use *type annotations* for this purpose. However, because the annotations clearly depend on run-time values (e.g., the contents of the pointer x in Figure 1), we have to use *dependent types*.

In particular, $\mathsf{HTT}_{\mathbf{cc}}$ features two type constructors which we use to provide annotations for side-effectful programs and for continuation objects. Intuitively, the type $\mathsf{SK}^*\ \{P\}\ A\ \{Q\}$ classifies programs with precondition P, postcondition Q and return value of type A. The type $\mathsf{Kont}^*\ \{R\}\{P\}\ A\ \{Q\}$ classifies continuation objects. R describes what holds when the continuation is captured by **callcc**, and is refered to as the *initial condition*. Precondition P describes what must hold of the heap when **abort**ing to the continuation object; thus immediately before the finalisation code is executed. postcondition Q describes the heap and the value obtained after the execution of the finalisation code, but before the actual jump. In both types, the assertions R, P and Q may depend on program values. Additionally, Q may depend on the dedicated variable $r{:}A$, naming the return value.

We employ the notation $[v_1{:}A_1, \ldots, v_n{:}A_n].\ \mathsf{SK}^*\ \{P\}\ A\ \{Q\}$, often omitting the types A_i, to specify that $v_1, \ldots, v_n$ are variables that may scope through P and Q (and similarly for R, P and Q in Kont^* types). In Hoare logic terminology, such variables are known as *logical*; they may appear in assertions, but not in the code. In first-order Hoare logics, logical variables have global scope and are used to relate the initial and ending states of a computation. In our setting, the scope of logical variables is *local* to the type in which they are bound. This is required in any Hoare logic for a language with procedures and recursion [24] such as $\mathsf{HTT}_{\mathbf{cc}}$, where a logical variable used in the specification of a recursive procedure, may have to be instantiated differently to satisfy the preconditions of different recursive calls.

$$
\begin{array}{l}
\mathbf{inc3}\,(x{:}\mathsf{ptr}) : [v].\ \mathsf{SK}^*\ \{x \mapsto v\}\ ()\ \{x \mapsto v+3\} \triangleq \\
\quad \mathbf{do}\ \ c \leftarrow \mathbf{callcc}_j \\
\qquad\qquad f : [v].\ \mathsf{Kont}^*\ \{j \in x \mapsto v\}\ \{x \mapsto v+1\}\ \Sigma{\bullet}\mathsf{SK}\,() \\
\qquad\qquad\qquad\qquad \{r.\ x \mapsto v+2 \curlywedge \\
\qquad\qquad\qquad\qquad\ \ \mathsf{spec}\, r \sqsubseteq (x \mapsto v+3, x \mapsto v+3)^*\}. \\
\qquad\qquad \mathbf{do}\,(\mathbf{ret}\ [\mathbf{abort}\ f\ \ (x\ :=\ !x+1\,;\ \mathbf{ret}\ [\mathbf{ret}\ ()])]) \\
\qquad\qquad : [v].\ \mathsf{SK}^*\ \ \{x \mapsto v \curlywedge j \in x \mapsto v\}\ \Sigma{\bullet}\mathsf{SK}\,() \\
\qquad\qquad\qquad\qquad \{r.\ x \mapsto v \curlywedge \mathsf{spec}\, r \sqsubseteq (x \mapsto v+1, \bot)^*\}; \\
\qquad x\ :=\ !x+1\,; \\
\qquad \mathbf{cmd}\ c.
\end{array}
$$

Figure 2. Specification of **inc3** via type annotations.

Additionally, we employ a *dependent record type* $\Sigma{\bullet}\mathsf{SK}\,A$, which packages a precondition and a postcondition together with a computation, and thus abstracts existentially over them. In other words, a value of type $\Sigma{\bullet}\mathsf{SK}\,A$ is a *structure* of the form $[P, Q, e]$, where $e : \mathsf{SK}^*\ \{P\}\ A\ \{Q\}$. As we show subsequently, values of this type will be used whenever we "nest" the monadic types, i.e. build computations that return other computations, which may potentially capture continuations. Given $c : \Sigma{\bullet}\mathsf{SK}\,A$, we will use $\mathsf{spec}\,c$ and $\mathbf{cmd}\ c$ to project the components when necessary:

$$
\begin{array}{l}
\mathsf{spec} : [P, Q, e] \mapsto (P, Q) \\
\mathbf{cmd} : [P, Q, e] \mapsto e
\end{array}
$$

We will abuse the notation and write $[e]$ instead of $[(P, Q), e]$, as the Hoare triple type of e—and hence its P and Q components—can usually be inferred, as we describe in Section 4. We will also use the symbol $\sqsubseteq$ to denote the usual Hoare ordering (i.e., pre-strengthening/post-weakening) on pairs (P, Q).

We now present in Figure 2 the fully annotated version of **inc3**, as it is written in $\mathsf{HTT}_{\mathbf{cc}}$. Apart from the obvious typing annotations and the explicit use of the aforementioned constructors and projections for $\Sigma{\bullet}\mathsf{SK}\,()$, there are additional syntactic elements that did not appear in Figure 1. We use the expression $\mathbf{do}\ e : \mathsf{SK}^*\ \{P\}\ A\ \{Q\}$ (potentially with logical variables) whenever we want to explicitly ascribe the specification (P, Q) to e, rather than use the tightest specification that the system infers for e. Such ascription will entail a proof obligation that (P, Q) is valid for e, as we explain in Section 4. When the type ascription is explicitly bound to a variable x, we write $x : \mathsf{SK}^*\ \{P\}\ A\ \{Q\} \triangleq \mathbf{do}\ e$.

We also make explicit that **callcc** captures the *current heap*, in addition to the current continuation, through the *heap* variable j that is bound by **callcc** and which we declare as an index (e.g., $\mathbf{callcc}_j$). The variable j scopes over the whole **callcc** body, including the type of the continuation object f. However, it is introduced strictly for purposes of specification, and we shall use it only in the annotations, but not in the executable code over which it scopes. The role of j is to relate the values of the various logical variables in its scope. In Figure 2, e.g., the assertion $j \in x \mapsto v$ appears in the type of the continuation object f and in the type

of the body of **callcc**, thus implying that the (distinct) logical variables named v in the two types, in fact denote the same value – that stored in x at the entry to **callcc**. In Figure 1 we used a single *global* logical variable v for this purpose, but, as explained, global logical variables do not scale to Hoare-style reasoning about procedural languages.

Figure 2 uses the constructor $[\_]$ twice. Once around $c_1 = \mathbf{abort}\ f\,(x\ :=\ !x+1\,;\ \mathbf{ret}\,[\mathbf{ret}\,()])$ and again around $c_2 = \mathbf{ret}\,()$, which is embedded inside c_1. As explained in Section 2.1 at different points of execution, both c_1 and c_2 are assigned to the variable c, and thus, all three must have the same type. Because the Hoare types of c_1 and c_2 actually differ in the pre- and postconditions, we employ $[\_]$ to abstract over the whole specifications, coercing c_1 and c_2 to $\Sigma\bullet\mathsf{SK}()$. The individual specifications of c_1 and c_2 are then re-established using the $\mathsf{spec}\ r$ projection in other program annotations.

For example, $[c_1]$ is the return value of **callcc**. The explicitly ascribed SK^* type states that $\mathsf{spec}\,r \sqsubseteq (x \mapsto v+1, \bot)^*$, exposing that c_1 performs a jump and should be executed only when x is incremented once.[2] On the other hand, $[c_2]$ is the value of the finalisation code of the **abort** to f in c_1. Hence, the variable r in the postcondition of f's type stands for $[c_2]$, and the formula $\mathsf{spec}\,r \sqsubseteq (x \mapsto v+3, x \mapsto v+3)^*$ in f's postcondition exposes that c_2 does not change the state, but should be executed only when x is incremented by 3. The types of f and the **callcc** body in Figure 2 explicitly provide the required information about the intended uses of the code bound to c at various execution points. Indeed, the postconditions of these two types are essentially the two disjuncts from line 3, Figure 1, with the formulas involving $\mathsf{spec}\,r$ replacing the Hoare triples over c. In this sense, the spec projection out of the record type $\Sigma\bullet\mathsf{SK}$ represents Hoare triples when they are *nested*, i.e. used within the assertions of other Hoare triples.

It is important, however, that the two disjuncts in line 3, Figure 1 are specified in two different places in **inc3** from Figure 2. In particular, the type of f provides the disjunct describing what happens when f is jumped to, whereas the inner type ascription provides the disjunct describing the normal return value of the **callcc** block. This pattern whereby f specifies only the jumping behaviour is characteristic of the algebraic **callcc** operator. On the other hand, as we shall see in Section 6, the annotations describing the non-jumping behaviour in the body of **callcc** can often be inferred (though in the case of **inc3** we had to explicitly ascribe them because the return value nests a jump). In the non-algebraic case, in our experience, f has to *always* be manually annotated with the full disjunction of the jumping and non-jumping cases, resulting in larger and more cumbersome annotations and proofs.

3. Notation, logical variables, large footprint

While logical variables are very useful in specifications, they are somewhat inconvenient to work with in the meta theory. The main hindrance is that premises of inference rules may contain Hoare triples with differing contexts of logical variables, which have to coalesce in some way into a logical context of the Hoare triple in the conclusion. Typically, simple conjoining of contexts is not what is wanted, as some sharing of variables is desired. But then, it becomes problematic how to specify just exactly which variables should be shared in the conclusion, and which should not. To circumvent the issue, in this section we introduce SK and Kont types which do not use logical variables at all, and show how SK^* and Kont^* types with logical variables from Section 2, become merely notational abbreviations.

As a first step, we introduce the type $\mathsf{SK}\ A\ (P,\ Q)$ of effectful computations where P is a precondition over heaps, as discussed in the previous section, but Q is a *binary* postcondition, ranging over the ending result of the program (of type A) and the initial and ending heaps, much as in VDM-style specifications [7, 30]. We use CIC-style notation to classify logical propositions by the type Prop, and represent predicates as functions into Prop.

Definition 1 (A-specs). *Given P : $\mathsf{pre}\mathbf{T}$ and Q : $\mathsf{post}\mathbf{T}\ A$ an A-specification, or A-spec, is a pair (P, Q) : $\mathsf{spec}\mathbf{T}\ A$ with:*

$$\begin{aligned}\mathsf{pre}\mathbf{T} &\triangleq \mathsf{heap} \to \mathsf{Prop}\\ \mathsf{post}\mathbf{T}\ A &\triangleq A \to \mathsf{heap} \to \mathsf{heap} \to \mathsf{Prop}\\ \mathsf{spec}\mathbf{T}\ A &\triangleq \mathsf{pre}\mathbf{T}\ \times \mathsf{post}\mathbf{T}\ A\end{aligned}$$

The notation $[\Delta].\,\mathsf{SK}^*\ \{P\}\ A\ \{Q\}$ from Section 2, with unary $\{P\}$ and $\{Q\}$ is an abbreviation, as we illustrate next.

Example 2. The type $[v].\,\mathsf{SK}^*\ \{x \mapsto v\}\ ()\ \{x \mapsto v+3\}$ of **inc3** from Figure 2, is an abbreviation of the specification:

$$\begin{aligned}&\mathsf{SK}\ ()\ (\lambda\, i.\, \exists\, v.\, i \in x \mapsto v,\\ &\qquad \lambda\, r\, i\, m.\, \forall\, v.\, i \in x \mapsto v \to m \in x \mapsto v+3)\end{aligned}$$

The SK type introduces an explicit quantification over v in the precondition to guarantee that **inc3** is safe to execute in the heap containing (only) the pointer x. The universal quantification in the postcondition expresses that upon termination, the value pointed to by x is incremented by 3. To express this property, the precondition $x \mapsto v$ has to be repeated as part of an implication in the binary postcondition, which is obviously cumbersome, thus motivating the following notation, generalising to a context Δ.[3]

Definition 3 (SK^*). *Given a type A, P : $\mathsf{pre}\mathbf{T}$, Q : $\mathsf{post}\mathbf{T}\ A$ the type notation $[\Delta].\,\mathsf{SK}^*\ \{P\}\ A\ \{Q\}$ is defined as:*

$$[\Delta].\,\mathsf{SK}^*\ \{P\}\ A\ \{Q\}\ \triangleq\ \mathsf{SK}\ A\ (P, Q)^*_\Delta$$

where $(P, Q)^*_\Delta$ is the A-spec defined as follows:

$$(P, Q)^*_\Delta \triangleq (\lambda\, i.\ \exists \Delta.\, i \in P,\ \lambda\, r\, i\, m.\ \forall \Delta.\, i \in P \to m \in Q)$$

The second step is to introduce the type $\mathsf{Kont}\ A\ R\ Q$ of continuation objects. R:$\mathsf{pre}\mathbf{T}$ is the initial condition, describing the heap at the point of a jump. Q:$\mathsf{post}\mathbf{T}\ A$ is a binary postcondition relating the initial heap with the ending heap and value of the finalisation code. The Kont type will always be in scope of a variable j denoting the heap at the point of continuation capture (the index of **callcc** in Figure 2); thus R and Q may depend on j as well.

We next show how the the type Kont^* from Section 2, is a notation over Kont, but first we need to generalise somewhat. As the following example illustrates, the Kont^* types actually require *two* different kinds of logical variables.

Example 4. Consider a continuation object which is captured when the heap $j = x \mapsto v$, can be jumped to only when x is incremented by *some* p, and whose finalisation code increments p further by 1. The scenario uses the program variable x and two logical variables, v and p. However, v and p clearly have different nature; v is implicitly universally quantified, while quantification over p is existential when describing the precondition for the jump, and universal in the description of the finalisation code. Thus, we put v and p into two different contexts, and describe the continuation object by the type

$$[v], \langle p\rangle.\,\mathsf{Kont}^*\ \{j \in x \mapsto v\}\{x \mapsto v+p\}\ A\ \{x \mapsto v+p+1\}$$

We refer to the two kinds of variables and contexts as *box* and *diamond* variables and contexts, respectively. In Figure 2 we only

[2] The reader can ignore the operation $(-)^*$ for now; it will be defined in Section 3.

[3] The notation also explains the operator $(-)^*$ (with the empty context Δ), that was used in Figure 2.

used the box contexts, and in general, when the diamond context is empty, we simply omit it. However, diamond context is not always empty, as we will illustrate in the **ping–pong** program in Section 6.2.

Definition 5 (Kont^*). *Let Δ and Γ be contexts of logical variables, and j be a distinguished heap variable, possibly occurring freely in R : preT, P : preT and Q : postT A. Then the notation $[\Delta], \langle\Gamma\rangle.\ \mathsf{Kont}^*\ \{R\}\{P\}\ A\ \{Q\}$, is an abbreviation for the following* Kont *type without logical variable contexts.*

$$[\Delta], \langle\Gamma\rangle.\ \mathsf{Kont}^*\ \{R\}\{P\}\ A\ \{Q\} \triangleq$$
$$\mathsf{Kont}\ A\ \ (\lambda\, i.\ \forall\Delta.\ R \to \exists\,\Gamma.\ i \in P)$$
$$(\lambda\, r\, m\, i.\ \forall\Delta.\ R \to \forall\Gamma.\ i \in P \to m \in Q)$$

All the variables in Δ are universally quantified in the notation, whereas the variables in Γ are existentially quantified in the initial condition, and universally in the postcondition.

We close the section by revisiting the issue of large footprint annotations. In Section 2, **inc3** could execute only in a heap with *exactly* the pointer x, and no others. We now explain how to specify **inc3** to admit execution in the presence of additional pointers. We will use a (box) logical variable of type heap, to describe the sub-heaps that *do not* contain x. For example, the more general annotation for **inc3** is given below.

$$\begin{array}{l}
\mathbf{inc3}\,(x{:}\mathsf{ptr}) : [v,h].\ \mathsf{SK}^*\ \ \{x \mapsto v * \mathsf{this}\,h\}\,()\\
\qquad\qquad\qquad\qquad\quad \{x \mapsto v+3 * \mathsf{this}\,h\} \triangleq\\
\ \ \mathbf{do}\ \ c \leftarrow \mathbf{callcc}_j\\
\qquad f : [v,h].\ \mathsf{Kont}^*\,\{j \in x \mapsto v * \mathsf{this}\,h\}\\
\qquad\qquad\qquad\quad \{x \mapsto v+1 * \mathsf{this}\,h\}\Sigma{\bullet}\mathsf{SK}\,()\\
\qquad\qquad\qquad\quad \{r.\ x \mapsto v+2 * \mathsf{this}\,h\\
\qquad\qquad\qquad\qquad \curlywedge\ \mathsf{spec}\,r \sqsubseteq (x \mapsto v+3 * \mathsf{this}\,h,\\
\qquad\qquad\qquad\qquad\qquad x \mapsto v+3 * \mathsf{this}\,h)^*\}.\\
\qquad \mathbf{do}\,(\mathbf{ret}\ [\mathbf{abort}\ f\ \ (x\ :=\ !x+1\,;\ \mathbf{ret}\ [\mathbf{ret}\ ()])])\\
\qquad : [v,h].\ \mathsf{SK}^*\ \{x \mapsto v * \mathsf{this}\,h \curlywedge j \in x \mapsto v * \mathsf{this}\,h\}\\
\qquad\qquad\qquad\quad \Sigma{\bullet}\mathsf{SK}\,()\\
\qquad\qquad\qquad\quad \{r.\ x \mapsto v * \mathsf{this}\,h\\
\qquad\qquad\qquad\qquad \curlywedge\ \mathsf{spec}\,r \sqsubseteq (x \mapsto v+1 * \mathsf{this}\,h, \bot)^*\};\\
\qquad x\ :=\ !x+1\,;\\
\qquad \mathbf{cmd}\ c.
\end{array}$$

The annotations introduce a logical variable h, the assertions $\mathsf{this}\,h$, and separating conjunction $*$, to name the part of the heap disjoint from the pointer x. The occurrence of the same $\mathsf{this}\,h$ in both the pre- and postcondition of **inc3**, specifies that **inc3** keeps this part of the heap invariant. Moreover, h is local to the Hoare triples in which it appears (unlike in first-order Hoare or Separation logic, where logical variables are global). Thus, a specification of **inc3** can be extended to a larger heap merely by instantiating h, rather than by means of a dedicated *frame rule* as in Separation logic. Because of the repeated occurrences of $\mathsf{this}\,h$ in the assertions, this style of annotation is a bit more verbose than in Separation logic, where the invariance of residual heaps is implicit. However, in practice, the extra logical variable did not affect our proofs. We discuss in Section 7 the reasons for needing large footprints, and some alternative designs.

4. Inference rules

We develop the semantics of $\mathsf{HTT_{cc}}$ using the Calculus of Inductive Constructions (CIC) as the meta logic. The choice provides us directly with a method to prototype $\mathsf{HTT_{cc}}$ as a shallow embedding in Coq, thus inheriting a number of useful constructs, such as

$$\begin{array}{ll}
\mathbf{ret} : & \Pi v{:}A.\,[h].\ \mathsf{SK}^*\ \{\mathsf{this}\,h\}\ A\ \{r.\ \mathsf{this}\,h \curlywedge r = v\}\\
\mathbf{alloc} : & \Pi v{:}A.\,[h].\ \mathsf{SK}^*\ \{\mathsf{this}\,h\}\ \mathsf{ptr}\ \{r.\ r \mapsto v * \mathsf{this}\,h\}\\
\mathbf{dealloc} : & \Pi x{:}\mathsf{ptr}.\,[B, v{:}B, h].\ \mathsf{SK}^*\ \{x \mapsto v * \mathsf{this}\,h\}\ ()\ \{\mathsf{this}\,h\}\\
:= : & \Pi x{:}\mathsf{ptr}.\,\Pi v{:}A.\,[B, w{:}B, h].\ \mathsf{SK}^*\ \ \{x \mapsto w * \mathsf{this}\,h\}\ ()\\
& \qquad\qquad\qquad\qquad\qquad\quad \{x \mapsto v * \mathsf{this}\,h\}\\
! : & \Pi x{:}\mathsf{ptr}.\,[v{:}A, h].\ \mathsf{SK}^*\ \ \{x \mapsto v * \mathsf{this}\,h\}\ A\\
& \qquad\qquad\qquad\qquad\qquad\quad \{r.\ x \mapsto v * \mathsf{this}\,h \curlywedge r = v\}
\end{array}$$

Figure 3. $\mathsf{HTT_{cc}}$ typing assignment for primitive commands.

dependent Π and Σ types, which we have already used in Section 2. For the sake of brevity, we omit the treatment of such standard constructs (it can be found in [27, 6]), and freely assume the standard typing rules and the various syntactic categories of CIC, such as, e.g., variable contexts. We only present our *impure* monadic extensions: the typing rules for the SK and Kont types, and related terms. In Section 5, we develop the semantic model for $\mathsf{HTT_{cc}}$, which we mechanised in Coq, to show the soundness of the extension.

The specific rules of $\mathsf{HTT_{cc}}$ are of two distinct kinds. The first kind consists of *typing rules*, which serve to infer the default program specifications (weakest precondition for memory safety, and strongest postcondition wrt. that precondition). The inference is important in practice because it minimises the amount of annotations that the user has to provide manually. The second kind consists of *structural lemmas* that formalise the reasoning about Hoare-ordering of specifications. As illustrated in Figure 2, such reasoning is needed in several situations: (1) We may explicitly need to ascribe a custom specification to a program, and this requires a proof that the default specification can be pre-weakened and post-strengthened into a desired one, and (2) We may use the relation $\sqsubseteq$, to explicitly declare that a $[-]$-abstracted command satisfies a predetermined specification. The two kinds of rules are discussed in Sections 4.1 and 4.2, respectively.

4.1 Typing rules

From here on, we use s and variants to range over specifications (P, Q), with $\mathsf{pre}\ s$ and $\mathsf{post}\ s$ projecting out the components.

Bind rule in Figure 4 perhaps best exemplifies the inference nature of our typing rules. Given programs e_1 and e_2 with specification s_1 and s_2, respectively, the rule infers the tightest specification for the sequential composition $x \leftarrow e_1; e_2$ as follows. Because the execution of the compositions starts with e_1, the inferred precondition must require that $\mathsf{pre}\ s_1$ holds of the initial heap i. After e_1 terminates with an intermediate value x and heap h satisfying $\mathsf{post}\ s_1\ x\ i\ h$, it must be that $\mathsf{pre}\,(s_2\ x)\ h$ so that e_2 is safe to run. The inferred postcondition declares the existence of an intermediate value x, and and a heap obtained after e_1 but before e_2, as per relational composition $\mathsf{post}\ s_1\ x \circ \mathsf{post}\,(s_2\ x)\ r$.

Bind, as well as the other rules in Figure 4, use SK and Kont types *without* logical variable contexts, which is why we introduced such types in Section 3 in the first place. Omitting logical variables facilitates specification inference, as it circumvents the issue of reconciling potentially different contexts of logical variables that may appear in the type for e_1 and the type for e_2.

Abort rule infers a precondition that has a dual role. The first conjunct $R\,i$ ensures that the continuation object f is **aborted** to only in heaps i for which the initial condition R is satisfied. The second conjunct $s \sqsubseteq (R \curlywedge \mathsf{this}\,i, Q)$ ensures that e is an appropriate finalisation code for f; that is, the specification s of e can be weakened into a precondition R and postcondition Q, as required by the type of f. Additionally, $\mathsf{this}\,i$ allows the proof of the weakening to

$$\frac{\Gamma \vdash e_1 : \mathsf{SK}\ A\ s_1 \qquad \Gamma, x : A \vdash e_2 : \mathsf{SK}\ B\ (s_2\ x)}{\Gamma \vdash x \leftarrow e_1; e_2 : \mathsf{SK}\ B\ (\lambda i.\ \mathsf{pre}\ s_1\ i \wedge \forall x\ h.\ \mathsf{post}\ s_1\ x\ i\ h \rightarrow \mathsf{pre}\ (s_2\ x)\ h, \lambda r\ i\ m.\ \exists x.(\mathsf{post}\ s_1\ x \circ \mathsf{post}\ (s_2\ x)\ r)\ i\ m)}\ \textsc{Bind}$$

$$\frac{\Gamma, j : \mathsf{heap}, f : \mathsf{Kont}\ A\ (R\ j)\ (Q\ j) \vdash e : \mathsf{SK}\ A\ (s\ j)}{\Gamma \vdash \mathbf{callcc}_j\ f\ .\ e : \mathsf{SK}\ A\ (\lambda i.\ \mathsf{pre}\ (s\ i)\ i,\ \lambda r\ i\ m.\ \mathsf{post}\ (s\ i)\ r\ i\ m \vee (R \circ Q\ i\ r)\ i\ m)}\ \textsc{CallCC}$$

$$\frac{\Gamma \vdash f : \mathsf{Kont}\ A\ R\ Q \qquad \Gamma \vdash e : \mathsf{SK}\ A\ s}{\Gamma \vdash \mathbf{abort}_B\ f\ e : \mathsf{SK}\ B\ (\lambda i.\ R\ i \wedge s \sqsubseteq (R \curlywedge \mathsf{this}\ i, Q),\ \lambda r\ i\ m.\ \mathsf{False})}\ \textsc{Abort}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{SK}\ A\ s_1 \quad s_1 \sqsubseteq s_2}{\Gamma \vdash \mathbf{do}\ e_1 : \mathsf{SK}\ A\ s_2}\ \textsc{Do} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \qquad \Gamma \vdash e_1 : \mathsf{SK}\ A\ s_1 \qquad \Gamma \vdash e_2 : \mathsf{SK}\ A\ s_2}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \mathsf{SK}\ A\ (\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2)}\ \textsc{If}$$

$$\frac{\Gamma \vdash f\ : (\Pi\, x{:}A.\ \mathsf{SK}\ (B\ x)\ (s\ x)) \rightarrow \Pi\, x{:}A.\ \mathsf{SK}\ (B\ x)\ (s\ x)}{\Gamma \vdash \mathbf{fix}\ f : \Pi\, x{:}A.\ \mathsf{SK}\ (B\ x)\ (s\ x)}\ \textsc{Fix}$$

Figure 4. $\mathsf{HTT}_{\mathbf{cc}}$ typing rules for specification inference.

exploit the knowledge that the heap in which the finalisation code executes is exactly i. The exact definition of $\sqsubseteq$ will be given in Section 4.2. Because **abort** does not return any values, its return type B is arbitrary and can be supplied by the user. We omit annotating this type in examples as it can be usually inferred from the context.

CALLCC rule in Figure 4, infers the specification for $\mathbf{callcc}\ f.\ e$. The premise of the rule introduces the heap variable j, which, as illustrated in Section 2, provides a common point for f and e to "synchronise" on, thereby fixing the values of various logical variables in relation to j. In the conclusion of the rule, j will be instantiated with the initial heap of **callcc**; that is, with the heap at the point of continuation capture.

The specification of f allows aborting to f only in heaps satisfying $R\ j$. After f's finalisation code terminates, the resulting heap and value satisfy $Q\ j$. If e has a specification s, then the tightest specification of $\mathbf{callcc}\ f.\ e$ can be inferred as follows. Because the execution of the whole command starts with e, the inferred precondition has to be derived out of e's precondition $\mathsf{pre}\ (s\ j)$. The unknown j is instantiated with the actual initial heap, to obtain the tightest precondition $\lambda i.\ \mathsf{pre}\ (s\ i)\ i$. The postcondition is a disjunction expressing that e may produce two different outcomes: it either aborts to f, or it does not. The left disjunct $\mathsf{post}\ (s\ i)\ r\ i\ m$ describes the non-aborting case; it simply equals the postcondition of e with j instantiated by i. In those cases when e actually aborts, the disjunct will be False, as it embeds the postcondition of the ABORT rule. The right disjunct is a relational composition $(R \circ Q\ i\ r)\ i\ m$ which describes the aborting case, as follows. In the aborting case, there exists a heap—call it h—that is current at the point of **abort**. Due to the specification of f, R must relate i and h. Additionally, m and r are obtained after the finalisation code of f is executed at h, and thus $Q\ i\ r$ must relate h and m as well.

The annotations specifying the continuation object (R and Q) appear in a negative position in the premise of CALLCC, and cannot be inferred automatically. In contrast, the specification s for e is generated by e's typing derivation. The property that *only* the aborting case has to be specified manually, differentiates the algebraic **callcc** from the standard, non-algebraic alternatives. A detailed comparison is presented in Section 7.

Other rules The rule DO implements a type ascription, requiring a proof that the specification s_1 can be weakened into s_2, as in the usual Hoare logic rule of consequence. The IF rule uses the type-level conditional, available in CIC, to compute the specification of a program-level conditional out of the types of the components. The FIX rule implements the usual typing for recursive procedures, requiring that an SK type be established for the procedure body, under a hypothesis that the recursive calls satisfy the same type. The primitive stateful commands have standard Separation logic specifications (Figure 3), except that they are extended to the large footprint idiom by naming the unused heap with the logical variable h.

Additional notation In the rest of the paper, we will use explicit names for the derived specifications in Figure 4. For example, we write $\mathsf{callcc\_s}\ R\ Q\ s$ for $(\lambda i.\ \mathsf{pre}\ (s\ i)\ i, \lambda r\ i\ m.\ \mathsf{post}\ (s\ i)\ r\ i\ m \vee (R \circ Q\ i\ r)\ i\ m)$, $\mathsf{abort\_s}\ R\ Q\ s$ for $(\lambda i.\ R\ i \wedge s \sqsubseteq (R \curlywedge \mathsf{this}\ i, Q), \lambda r\ i\ m.\ \mathsf{False})$, and similarly for $\mathsf{bind\_s}$, $\mathsf{read\_s}$, etc.

4.2 Structural lemmas and symbolic evaluation

The Hoare ordering on two specifications s_1 and s_2 is defined as:

$$s_1 \sqsubseteq s_2 \triangleq \forall i.\ \mathsf{pre}\ s_2\ i \rightarrow \mathsf{verify}\ i\ s_1\ (\lambda r{:}A\ m.\ \mathsf{post}\ s_2\ y\ i\ m)$$

where for any $\kappa : A \rightarrow \mathsf{heap} \rightarrow \mathsf{prop}$:

$$\mathsf{verify}\ i\ s\ \kappa \triangleq \mathsf{pre}\ s\ i \wedge \forall r\ m.\ \mathsf{post}\ s\ r\ i\ m \rightarrow \kappa\ r\ m$$

The definition of $s_1 \sqsubseteq s_2$ states that $\mathsf{pre}\ s_1$ weakens into $\mathsf{pre}\ s_2$ and $\mathsf{post}\ s_2$ strengthens into $\mathsf{post}\ s_1$, as in the Hoare logic rule of consequence. It is split into two stages in order to exploit the hypothetical reasoning of CIC and Coq. In practice, the hypothesis $\mathsf{pre}\ s_2\ i$ will always move into the hypothesis context of Coq, leaving verify to describe the remaining proof goal.

The $\mathsf{HTT}_{\mathbf{cc}}$ proof obligations arise directly from the side condition about Hoare ordering in the rule DO in Figure 4, when the user wants to ascribe a desired specification to a program. In the practical work with $\mathsf{HTT}_{\mathbf{cc}}$, the proof obligations are discharged by applying a number of carefully crafted lemmas about verify that implement verification by *symbolic evaluation*. We illustrate the process here, and discuss the relationship to algebraicity of control operators in Section 4.3.

For instance, let e_1 and e_2 be computations with specifications s_1 and $s_2\ x$, respectively. The inferred specification for $x \leftarrow e_1; e_2$ is $\mathsf{bind\_s}\ s_1\ s_2$. An $\mathsf{HTT}_{\mathbf{cc}}$ proof obligation establishing that in some heap i, the execution of the sequential composition produces a heap and a value satisfying κ, will have the form:

$$\mathsf{verify}\ i\ (\mathsf{bind\_s}\ s_1\ s_2)\ \kappa$$

The idea of symbolic evaluation is to discharge such a goal as follows. It suffices to show that verifying a composite B-spec $(\mathsf{bind\_s}\ s_1\ s_2)$ can be reduced to verifying s_1 against a κ', where κ' itself involves verifying s_2 against κ. The process is iterated as long as s_2 contains sequential compositions, and can be seen as a

sequence of applications of the following lemmas in $\mathsf{HTT_{cc}}$:

$$\begin{aligned}\textsc{Step} &: \mathsf{verify}\ i\ s_1(\lambda\, y\, m.\ \mathsf{verify}\ m\ (s_2\ y)\ \kappa) \to \\ &\quad \mathsf{verify}\ i\ (\mathsf{bind\_s}\ s_1\ s_2)\ \kappa \\ \textsc{ValDo} &: \mathsf{pre}\ s\ i \to (\forall\, x\, i\, m.\ \mathsf{post}\ s\, x\, i\, m \to \kappa\, x\, m) \to \\ &\quad \mathsf{verify}\ i\ s\ \kappa\end{aligned}$$

The STEP lemma implements the iterative step, and VALDO lemma applies in the end, when there are no outstanding sequential compositions to be stepped through.

The VALDO lemma may be specialised to streamline the symbolic evaluation for specific commands. For example, let f : $\mathsf{Kont}\ A\, R\, Q$ and e : $\mathsf{SK}\ A\, s$. The inferred spec for $\mathbf{abort}\ f\ e$ is $\mathsf{abort\_s}\ R\, Q\ s = (\lambda\, i.\ R\, i \wedge s \sqsubseteq (R, Q), \lambda\, r\, i\, m.\ \mathsf{False})$. Taking this spec for s in VALDO, after some simplification, we obtain:

$$\begin{aligned}\textsc{ValAbort} : R\, i \to \mathsf{verify}\ i\ s\ (\lambda r.\ \lambda m.\ Q\, r\, i\, m) \to \\ \mathsf{verify}\ i\ (\mathsf{abort\_s}\ R\, Q\ s)\ \kappa\end{aligned}$$

In other words, to verify $\mathbf{abort}$ to f, we need to show that R holds of the current heap, and that the supplied finalisation code satisfies Q after running in i.

In case $e_1 = \mathbf{callcc}_j\ f.\, e$, the inferred spec is $\mathsf{callcc\_s}\ R\, Q\ s = (\lambda\, i.\ \ \mathsf{pre}\ (s\, i)\, i, \lambda\, r\, i\, m.\ \ \mathsf{post}\ (s\, i)\, r\, i\, m\ \vee\ (R \circ Q\, i\, r)\ \ i\, m)$, for some j and f : $\mathsf{Kont}\ A\ (R\, j)\ (Q\, j)$ and e : $\mathsf{SK}\ A\ (s\, j)$. Taking this spec for s, and after some rearrangement of the disjunction in the postcondition of callcc_s, VALDO can be specialised into the following lemma.

$$\begin{aligned}\textsc{ValCC} : \mathsf{verify}\, i\, (s\ i)\, \kappa \to \\ (\forall\, r{:}A\, m.\ (R \circ (Q\, i\, r))\, i\, m \to \kappa\, r\, m) \to \\ \mathsf{verify}\, i\, (\mathsf{callcc\_s}\ R\, Q\, s)\, \kappa\end{aligned}$$

The first hypothesis corresponds to the case when e does not abort. In that case, the goal reduces to verifying $(s\ i)$ against κ. The second hypothesis corresponds to the aborting case. In that case e produces an ending heap m and value x satisfying $(R \circ (Q\ i))\, x\, i\, m$; that is e first reaches an aborting heap out of i (predicate R), and then executes the finalisation code $(Q\ \ i)$. Then we just need to prove that κ holds after the execution of the finalisation code.

Specialised symbolic execution lemmas can be proved for all other primitive commands. Furthermore, because verify is an ordinary logical definition, users can establish such lemmas for their own programs as well, directly in the logic.

Example 6. We can implement the usual $\mathbf{throw}$ command, as an $\mathbf{abort}$ with an immediately-returning finalisation code. Given f : $\mathsf{Kont}\ A\, R\, Q$ and a value $v{:}A$, we define:

$$\mathbf{throw}\ g\ v \triangleq \mathbf{abort}\ f\ (\mathbf{ret}\ v)$$

The inferred specification $\mathsf{throw\_s}\ R\, Q\ v = \mathsf{abort\_s}\ R\, Q\ (\mathsf{ret\_s}\ v)$ can be proved to satisfy a streamlined version of VALABORT, which exploits the trivial nature of the finalisation code to simplify one of the hypotheses.

$$\textsc{ValThrow} : R\, i \to Q\, v\, i\, i \to \mathsf{verify}\ i\ (\mathsf{throw\_s}\ R\, Q\ v)\ \kappa$$

4.3 Algebraicity at the level of specifications

In this section, we show that the algebraicity property of $\mathbf{callcc}$ can be expressed as a lemma over specifications, similar to the symbolic evaluation lemmas from the previous section. This lemma, which we call ALGCC, expresses that we can commute the specification forms bind_s and callcc_s, thus lifting to the level of specifications the algebraicity equation (1) from Section 2. However, stating the ALGCC lemma requires generalising somewhat the definition of $\mathsf{callcc\_s}\ R\, Q\, s_1$. We first introduce the generalised definition, which we rename $\mathsf{algcc\_s}\ R\, s_0\, s_1$, and then describe the intuition behind it.

Consider a program of the form $x \leftarrow (\mathbf{callcc}_j\ f.\, e_1); e_2$, where f : $\mathsf{Kont}\ A\ (R\, j)\ (Q\, j)$, e_1 : $\mathsf{SK}\ A\ (s_1\, j)$, and $x{:}A \vdash e_2$: $\mathsf{SK}\ B\ (s_2 x)$. Then, side-by-side, the two definitions look like:

$$\begin{aligned}\mathsf{callcc\_s}\ R\, Q\, s_1 &= (\lambda\, i.\ \mathsf{pre}\ (s_1\, i)\, i \\ &\qquad \lambda\, r\, i\, m.\ \mathsf{post}\ (s_1\, i)\, r\, i\, m \vee (R \circ Q\, i\, r)\, i\, m) \\ \mathsf{algcc\_s}\ R\, s_0\, s_1 &= (\lambda\, i.\ \mathsf{pre}\ (s_1\, i)\, i \wedge \forall\, h.R\, i\, h \to \mathsf{pre}\ (s_0\, i)\, h, \\ &\qquad \lambda\, r\, i\, m.\ \mathsf{post}\ (s_1\, i)\, r\, i\, m \\ &\qquad\quad \vee (R \circ \mathsf{post}\ (s_0\, i)\, r)\, i\, m)\end{aligned}$$

The main difference is that where callcc_s uses only the postcondition Q (abstracting over j) to specify the finalisation code, algcc_s takes a full specification s_0 (also abstracting over j), which includes a precondition as well. In callcc_s, the precondition for the finalisation code is assumed to be the trivially true heap predicate, and we can prove the equation

$$\mathsf{callcc\_s}\ R\, Q\, s_1 = \mathsf{algcc\_s}\ R\, (\lambda\, j.\ (\top, Q\ j))\ s_1.$$

Intuitively, callcc_s could use the trivial precondition for the finalisation code, because R already describes what holds of the heap at the point of abort, and this is the same heap in which the finalisation code executes. However, the main property of algebraic commutation is that it changes the finalisation code by extending it with e_2, though it does not change the point of $\mathbf{abort}$ing. The new definition algcc_s thus divorces R and pre s_0, so that the algebraicity lemma can express that R remains fixed, while pre s_0 changes. The changes to pre s_0 cannot be arbitrary, however, as the finalisation code always executes in a heap in which $\mathbf{abort}$ is called. Thus, the precondition of algcc_s includes a conjunct that R implies pre s_0 for every initial heap i, and aborting heap h.

We can now state the algebraicity lemma for $\mathbf{callcc}$, with the omitted proof included in our Coq files.

$$\begin{aligned}\textsc{AlgCC} : \mathsf{verify}\, i\, (\mathsf{bind\_s}\ (\mathsf{algcc\_s}\ R\, s_0\, s_1)\, s_2)\ \kappa \quad \longleftrightarrow \\ \mathsf{verify}\, i\, (\mathsf{algcc\_s}\ R\ (\lambda\, j.\ \mathsf{bind\_s}\ (s_0\ j)\ s_2) \\ (\lambda\, j.\ \mathsf{bind\_s}\ (s_1\, j)\, s_2))\ \kappa\end{aligned}$$

5. Denotational semantics

In this section we present the semantics of $\mathsf{HTT_{cc}}$ as a shallow embedding into CIC. That is, we provide a semantic interpretation function $[\![-]\!]$ that maps $\mathsf{HTT_{cc}}$ types SK and Kont into types defined in CIC, but acts as identity on all the other types inherited from CIC, such as nat or heap. Similarly, the interpretation of terms maps the monadic constructs such as $\mathbf{callcc}$, $\mathbf{abort}$, etc., into CIC-terms, while acting as an identity on all the other terms inherited from CIC. The interpretation extends homomorphically to variable contexts as well.

5.1 Semantics of types

As customary in the case of control operators, our denotational semantics is parameterised wrt. the type X of return results for the continuations. We further require that X is a *complete lattice*, thus providing us with means to model the fixed point combinator in CIC, using the Knaster-Tarski theorem.

Definition 7. *Given a type A, and predicates P : preT and Q : postT A, the types* $\mathsf{SK}\ A\ (P,\ Q)$ *and* $\mathsf{Kont}\ A\, P\, Q$ *are interpreted as follows.*

$$\begin{aligned}&[\![\mathsf{SK}\ A\ (P,\ Q)]\!] \triangleq \\ &\quad \Pi i{:}\mathsf{heap}.\ [\![P]\!]\ i \to (\Pi r{:}[\![A]\!].\ \Pi m{:}\mathsf{heap}.\ [\![Q]\!]\, r\, i\, m \to X) \to X \\ &[\![\mathsf{Kont}\ A\, P\, Q]\!] \triangleq [\![\mathsf{SK}\ A\ (P,\ Q)]\!] \to \Pi j{:}\mathsf{heap}.\ [\![P]\!]\ j \to X\end{aligned}$$

A computation of SK type takes a heap i satisfying the precondition P, and a *continuation* requiring the postcondition Q as a precondition. Intuitively, the continuation applies to the ending value and heap of the computation, to produce a result in X. As the latter type is a complete lattice, the possible results include divergence, denoted by the bottom element of the lattice. We don't

model the faulting behaviour such as type or memory errors (e.g., de-referencing a dangling pointer), but instead rely on the proofs of $[\![P]\!]\,i$ and $[\![Q]\!]\,r\,i\,m$ to statically ensure that a computation executes only in heaps and continuations for which such errors do not occur. In this sense, well-typed (i.e., well-specified) computations in $\mathsf{HTT}_{\mathbf{cc}}$ do not fault, as usual for type systems and for *fault-avoiding* Hoare logics such as Separation logic.

A further useful intuition about our model may be gained if one erases the dependencies on P and Q in Definition 7. This results in the following informal equations:

$$\begin{array}{l}[\![\mathsf{SK}\,A]\!] \;\; = \mathsf{heap} \to ([\![A]\!] \to \mathsf{heap} \to X) \to X \\ [\![\mathsf{Kont}\,A]\!] = [\![\mathsf{SK}\,A]\!] \to \mathsf{heap} \to X\end{array}$$

The first equation shows that the $\mathsf{SK}\,A$ type is essentially the standard *state-passing continuation monad*. The second equation shows that a continuation object semantically requires two arguments: an explicit finalisation code (the $\mathsf{SK}\,A$ type), and a heap. Of course, the finalisation code is explicitly provided by the program as an argument of abort. Importantly, the heap argument is *implicitly* supplied by the denotation of $\mathbf{abort}$ as the heap current at the point of $\mathbf{abort}$ing to f.

Of course, the parametrisation with finalisation code is what makes our control operators algebraic, and is directly inspired by Jaskelioff [22]. On the other hand, the further heap argument makes the $\mathsf{Kont}\,A$ type implement *non-rollbacking* continuations. In contrast, the algebraic $\mathbf{callcc}$ presented by Jaskelioff for a state and continuations monad does roll-back the state to the one captured together with the current continuation.

5.2 Semantics of computations

For the sake of brevity, we present only the denotational semantics of the control operators $\mathbf{callcc}$ and $\mathbf{abort}$. Moreover, we illustrate only the simplified setting where the dependencies on P and Q are erased from the types (as in the above informal equations), and the dependencies on the proofs of $[\![P]\!]\,i$ and $[\![Q]\!]\,r\,i\,m$ are erased from the terms. The full dependency-respecting denotations for all the monadic commands from Figures 4 and 3 are implemented in the companion Coq files.

Definition 8 ($\mathbf{callcc}\ f.\,e$)**.** *Given* $f : [\![Kont\ A]\!]$ *and* $e : [\![\mathsf{SK}\,A]\!]$, *the denotation of* $\mathbf{callcc}\ f.\,e$ *of type* $[\![\mathsf{SK}\,A]\!]$ *is defined as:* [4]

$$\begin{array}{l}[\![\mathbf{callcc}]\!] \quad\;\; : ([\![\mathsf{Kont}A]\!] \to [\![\mathsf{SK}\,A]\!]) \to [\![\mathsf{SK}\,A]\!] \\ [\![\mathbf{callcc}\ f.\,e]\!] \triangleq\; \lambda i : \mathsf{heap}.\,\lambda k : [\![A]\!] \to \mathsf{heap} \to X. \\ \qquad\qquad [\lambda c : [\![\mathsf{SK}\,A]\!].\,\lambda h : \mathsf{heap}.\,c\;h\;k/f][\![e]\!]\;i\;k\end{array}$$

Intuitively, executing $\mathbf{callcc}\ f.\,e$ corresponds to applying the denotation to the initial (i.e., captured) heap i and continuation k. The body e is executed using the same heap and continuation. However, first the variable $f : [\![\mathsf{Kont}\,A]\!]$ is bound to a continuation object that, when supplied the finalisation code c and a heap h that is current at the point of aborting to f, executes c in h passing the control (i.e., jumping) to k.

Definition 9 ($\mathbf{abort}_B\ f\ e$)**.** *Given* $f : [\![\mathsf{Kont}\,A]\!]$ *and* $e : [\![\mathsf{SK}\,A]\!]$, *the denotation of* $\mathbf{abort}_B\ f\ e$ *of type* $[\![\mathsf{SK}\,B]\!]$ *is given by:*

$$\begin{array}{l}[\![\mathbf{abort}_B]\!] \quad\;\; : [\![\mathsf{Kont}\,A]\!] \to [\![\mathsf{SK}\,A]\!] \to [\![\mathsf{SK}\,B]\!] \\ [\![\mathbf{abort}_B\ f\ e]\!] \triangleq \lambda i : \mathsf{heap}.\,\lambda k : [\![B]\!] \to \mathsf{heap} \to X.\,[\![f]\!]\;[\![e]\!]\;i\end{array}$$

Theorem 10 (Soundness)**.** *If* $\Gamma \vdash e : A$ *then* $[\![\Gamma]\!] \vdash_{CIC} [\![e]\!] : [\![A]\!]$.

Proof. The proof is by induction on the structure of e. The interesting cases are when e is one of the monadic commands (correspondingly, when A is an SK type), as in all other cases the semantic function is trivial. When e is a monadic command, the soundness proof for the command is intertwined with the definition of the denotation of e. For example, in the case of $e = \mathbf{callcc}_j\ f.\,e$ the denotation will involve parametrization on the proofs of $[\![P]\!]\,i$ and $[\![Q]\!]\,r\,i\,m$, for the appropriate P and Q, that we have simplified away in our discussion. Such proof parameters are used in the denotations to build larger proofs on-the-fly, as necessary to make the various sub-terms of the denotation typecheck, until ultimately the whole denotation term typechecks wrt. the specification given in Figure 4. Similar considerations apply for other monadic terms as well. One exception is the fixed point construct $\mathbf{fix}\ f$, whose soundness is proved by an appeal to Knaster-Tarski theorem over the monotone completion of f. The Knaster-Tarski theorem applies because $[\![ST\,A\,(P,Q)]\!]$ is a complete lattice, being defined as a function space into a complete lattice X. We have mechanised all the steps of the proof in our Coq files. □

[4] Omitting the index j to $\mathbf{callcc}$, which may only appear in the erased dependencies.

```
1. rember-up-to-last (x : A) (xs : list A) :
     [h]. SK* {this h} (list A) {r. this h ↛ r = rember x xs [ ]} ≜
2.   do (callcc_j
          exit : [h]. Kont* {j ∈ this h}
                            {if x ∈ xs then this h else ⊥}
                            (list A)
                            {r. this h ↛ r = rember x xs [ ]}.
3.        fix (λ f : remberT. λ ys : list A.
4.             if ys is y :: ys' then
5.                  zs ← f ys';
6.                  if x == y then throw exit zs
7.                  else ret (y::zs)
8.             else ret [ ]) xs)
```

where

```
remberT ≜ Πys : list A.
  SK* {this j ↛ ∃ps. xs = ps++ys} (list A)
      {r. this j ↛ r ∈ remberP x ys [ ]}

remberP x xs acc ≜ if xs is y :: ys' then
                       if x == y then ∅
                       else remberP x ys' (acc++[y])
                   else {ps | ps = acc}
```

Figure 5. rember-up-to-last in $\mathsf{HTT}_{\mathbf{cc}}$.

6. A short verification survey

We present the verification of two examples exhibiting non-trivial patterns for programming with continuations. The first example, rember-up-to-last, illustrates *downwards or exit* continuations, whereby a jump is used to escape early from a recursive call, whilst discarding suspended computations. The second example, ping–pong, illustrates *upwards* continuations or *unstructured* loop, where the executions of a captured continuation are interleaved with user code, thereby mimicking the cooperating behaviour of (a simplified version of) coroutines. In the companion files [12], we verify other examples as well, such as escaping from infinite loops and using continuations to implement error handlers.

```
1. rember (x : A) (xs acc : list A) : list A ≜
2.         if xs is y::ys' then
3.             if x == y then rember x ys' [ ]
4.             else rember x ys' (acc++[y])
5.         else acc.
```

Figure 6. Purely functional rember.

6.1 rember-up-to-last

Let A be a type supporting decidable equality; i.e., there exists a function $==: A \to A \to \mathsf{bool}$. Given $x : A$ and a list $xs : \mathsf{list}\, A$, rember-up-to-last $x\, xs$, returns the ending segment of xs after the last occurrence of x; if x does not occur in xs, the whole xs is returned [18, p. 55]. For example, if $xs = [23, 16, 42, 4, 42, 8, 15, 16]$, then rember-up-to-last $4\ xs = [42, 8, 15, 16]$, rember-up-to-last $16\ xs = [\,]$ and rember-up-to-last $42\, xs = [8, 15, 16]$. However, rember-up-to-last $7\ xs = xs$.

Figure 5 implements rember-up-to-last in $\mathsf{HTT_{cc}}$, following the ML implementation by Thielecke [47]. For simplicity, we use purely-functional lists instead of imperative, singly- or doubly-linked lists. The implementation works as follows. In line 2, it captures the continuation with **callcc**. Then, it recurses over the input list xs, searching for x (lines 3–5), rebuilding the input list on the way back (line 7). If x is found (line 6), it jumps out of the loop, by **throw**ing to the captured continuation (Example 6). The returned value zs is the list rebuilt so far, while the outstanding iterations of the loop intended to further rebuild the list, are cancelled.

We also develop a purely-functional version rember (Figure 6) by induction on xs. rember does not use **callcc**, but instead relies on *tail recursion* and the accumulator acc to keep track of the currently rebuilt list. Of course, the implementation with jumps is preferable from the efficiency standpoint, but we require the pure rember in order to *specify* rember-up-to-last.

We now analyse the type annotations in Figure 5. Those for rember-up-to-last (line 1) are quite intuitive. The function can run in an arbitrary heap h (this h in the precondition). It leaves the heap unchanged (this h in the postcondition), and the return result r is the same as running the tail-recursive rember with the empty initial accumulator. The continuation object $exit$ (line 2) is specified as follows. The initial condition exposes that the captured heap j equals h. A jump to exit may occur only when the precondition is satisfied; in this case, only when $x \in xs$. The postcondition states that the heap is unchanged, and the return value equals running rember, as expected.

The most interesting annotations is the *loop invariant* remberT provided as the type of the recursive function f. The precondition states that f is only ever applied to the tail ys of xs; hence xs can be partitioned as $ps{+\!\!+}ys$. The partitioning is made explicit in the precondition, as it will be required when proving that we are throwing the correct ending segment in line 6. In the postcondition, remberT has to state that f produces the correct result. Importantly, however, it cannot use the helper function rember! rember requires an accumulator argument, but as f itself *is not* tail recursive, it is not clear which value to supply for the accumulator. Unlike in the specifications of rember-up-to-last and exit, it's incorrect to use $[\,]$, as that does not reflect the looping behaviour. It's also incorrect to existentially abstract over the accumulator, as that produces too weak a property which then does not imply the postcondition of rember-up-to-last, where the accumulator is $[\,]$.

The workaround is to define a helper function remberP, which is similar to rember, but records when the jumps in f appear, as follows. remberP returns either an empty *set* of values, to signalise a jump, or a singleton set, with the correct value, in the non-jumping case. More precisely, we have the following lemma:

$$\mathsf{rmb\_rmbP} : r \in \mathsf{remberP}\ x\ xs\ acc\ \to r = \mathsf{rember}\ x\ xs\ acc.$$

Then, the loop invariant remberT can assert that the return value r is always in the set defined by remberP. In the case of a jump, this property is evidently false, since the set is empty. But, in such a case, the program behaviour is described by the annotation on exit anyway, and the loop invariant need not bother describing it again.

Figure 7 presents a proof outline for rember-up-to-last $x\ xs$. The trivially true assertion $\exists h.\, \mathsf{this}\, h$ in line 1 corresponds to unfolding the notation for the type SK^* with a precondition this h, and a logical variable h. In line 3, the current heap h is captured into the variable j, corresponding to substituting h for j in the rule BNDCC, Section 4. In the proof outline, we cannot represent the substitution because j appears in the rest of the code, so instead we equate j with h in the assertion. Line 6 starts the verification of the loop; it shows that the precondition of the loop invariant remberT holds; the heap is unchanged wrt. the captured heap, and there exists a partition $xs = ps{+\!\!+}ys$. One critical point in the proof is Line 13, where we need to establish $zs = \mathsf{rember}\ x\ xs\ [\,]$, which is the precondition for **throw**. This property can be proved out of the partitioning $xs = ps{+\!\!+}(x :: ys')$ and $zs \in \mathsf{remberP}\ x\ ys'\ [\,]$ available in Line 12, by using rmb_rmbP and additional two helper

```
1. {∃h. this h}
2. do ( callcc_j exit .
3. {∃h. this h ⅄ j = h}
4. {this j ⅄ xs = [ ]++xs}
5.   fix (λ f ys.
6. {∃ps. this j ⅄ xs = ps++ys}
7.                 if ys is y :: ys' then
8. {∃ps. this j ⅄ xs = ps++(y:: ys')}
9.                     zs ← f ys';
10. {∃ps. this j ⅄ xs = ps++(y:: ys') ⅄ zs ∈ remberP x ys' [ ]}
11.                 if x == y then
12. {∃ps. this j ⅄ xs = ps++(x::ys') ⅄ zs ∈ remberP x ys' [ ]}
13. {this j ⅄ x ∈ xs ⅄ zs = rember x xs [ ]}
14.                     throw exit zs
15. {⊥}
16.                 else ret (y::zs)
17. ∃ps. this j ⅄ xs = ps++(y:: ys')
                ⅄ y::zs ∈ remberP x (y::ys') [ ]}
18.             else ret [ ]
19. {∃ps. this j ⅄ xs = ps ⅄ [ ] ∈ remberP x [ ] [ ]}
20.                     ) xs)
21. {this j ⅄ r ∈ remberP x xs [ ]}
22. {∃h. this h ⅄ (r ∈ remberP x xs [ ] ϒ
                  (x ∈ xs ⅄ r = rember x xs [ ]))}
23. {∃h. this h ⅄ r = rember x xs [ ]}
```

Figure 7. Proof outline for rember-up-to-last $x\ xs$.

lemmas (though we elide the derivation here):

$$\mathsf{rmb\_cat}: \quad \mathsf{rember}\ x\ (ps +\!\!+ ys)\ acc = \mathsf{rember}\ ys\ (\mathsf{rember}\ x\ acc\ ps)$$

$$\mathsf{rmb\_in}: \mathsf{rember}\ x\ xs\ acc = \mathsf{if}\ x \in xs\ \mathsf{then}\ \mathsf{rember}\ x\ xs\ [\,]\ \mathsf{else}\ (acc +\!\!+ xs)$$

Another critical point is line 17, where we need to prove $y :: zs \in \mathsf{rememberP}\ x\ (y :: ys')\ [\,]$, required to establish the postcondition of the loop. The property is proved out of $zs \in \mathsf{rememberP}\ x\ ys'\ [\,]$ available in line 10, after unfolding the definition of rememberP once, and using the following lemma about rememberP:

$$zs \in \mathsf{rememberP}\ x\ ys\ acc \to y{::}zs \in \mathsf{rememberP}\ x\ ys\ (y{::}acc).$$

Line 19 describes what holds in the else branch of the main conditional in the loop, and line 21 establishes that the loop invariant holds at the end of the loop. Line 21 is a common weakening of lines 15, 17 and 19. Line 22 includes a disjunction, showing that the line can be reached by a normal termination of the loop (line 21), or by a jump to exit. Line 23 is obtained out of line 22 by applying rmb_rmbP, and establishes the specified postcondition of rember-up-to-last.

6.2 Ping-Pong cooperation

In Section 2 we presented **inc3**, which used a closure to capture a continuation and execute it twice. Here, we generalise the idea to n calls to **abort**, thus iterating n times the captured continuation. The result is an interleaving between the captured continuation and the finalisation code in the closure, which creates a cooperation pattern resembling that of coroutines [47] – albeit one without the full power of coroutine *fork* or *yield* operations, which further require storing continuations into the heap [40, 35]. Unlike **inc3**, where both the captured continuation and the finalisation code executed the same code, we will have different computations for the captured continuation, ping, and for the finalisation code in the closure, pong, and use the pre- and postconditions to show that these are interleaved in the evaluation of ping–pong. Since our motivation is to verify the cooperation pattern, rather than ping and pong *per se*, we give a trivial implementation for the latter functions: $\mathsf{ping} \triangleq \mathsf{incr}\ x$ and $\mathsf{pong} \triangleq \mathsf{incr}\ y$, where:

$$\begin{aligned}&\mathsf{incr}\ z : [v\ h].\ \mathsf{SK}^*\ \{z \mapsto v * \mathsf{this}\ h\}\ (\,)\ \{z \mapsto v+1 * \mathsf{this}\ h\}\\ &\quad \triangleq \mathbf{do}\ (v \leftarrow !x;\ x := v+1\,)\end{aligned}$$

Figure 8 presents ping–pong, whose structure is close to that of **inc3**. In line 2, **callcc** captures the continuation corresponding to the rest of the program $\lambda c.\ \mathsf{ping}; \mathbf{cmd}\ c$ and binds it to the continuation object k. The body of **callcc** returns a function f defined recursively on n. If n is non-zero (line 5), f returns a closure which aborts to the captured continuation with the finalisation code consisting of pong followed by the recursive call to f. In the zero case, f returns the computation pong. The result of this recursive function is bound to c in the sequel. When $n > 0$, c will be bound to a closure with n calls to **abort** nested in the finalisation code:

$$[\mathbf{abort}\ \underbrace{k\ (\mathsf{pong}; \mathbf{ret}\ [\mathbf{abort}\ k\ (\cdots\ \mathbf{abort}\ k}_{n \text{ calls to } \mathbf{abort}\ k}\ (\mathbf{ret}\ [\mathsf{pong}]))])]$$

After **callcc**, ping is evaluated to increment x, and then c is evaluated. If $n = 0$, and thus c is bound to [pong], the value at y is incremented an the function terminates. If $n > 0$, the *outermost* **abort** in the closure above is executed, thus running pong as the finalisation code and passing the rest of the nested computations in the closure, bound to c, to the captured continuation. This results in a backward jump to line 7. The loop continues until the closure is consumed in full and the function terminates. As a result ping and pong are interleaved $n+1$ times.

1. $\mathsf{ping–pong}\ (n : \mathsf{nat}) :$
   $[v\ w\ h].\ \mathsf{SK}^*\{x \mapsto v * y \mapsto w * \mathsf{this}\ h\}(\,)$
   $\{x \mapsto v+n+1 * y \mapsto w+n+1 * \mathsf{this}\ h\} \triangleq$
2. $\mathbf{do}\ (c \leftarrow \mathbf{callcc}_j$
   $k\ :\ [v\ w\ h]\langle p\rangle.$
   $\mathsf{Kont}^*\ \{j \in x \mapsto v * y \mapsto w * \mathsf{this}\ h\}$
   $\{\mathsf{if}\ n \neq 0\ \mathsf{then}$
   $x \mapsto v+p+1 * y \mapsto w+p * \mathsf{this}\ h$
   $\mathsf{else}\ \bot\}\ \Sigma\bullet\mathsf{SK}\ (\,)$
   $\{r.\ x \mapsto v+p+1$
   $* y \mapsto w+p+1 * \mathsf{this}\ h$
   $\curlywedge\ \mathsf{spec}\ r \sqsubseteq (x \mapsto v+p+2 *$
   $y \mapsto w+p+1 * \mathsf{this}\ h,$
   $\mathsf{if}\ p+1 = n\ \mathsf{then}$
   $x \mapsto v+n+1\ *$
   $y \mapsto w+n+1 * \mathsf{this}\ h$
   $\mathsf{else}\ \bot)^*\}.$
3. $\quad\quad \mathbf{fix}\ (\lambda f :\ \mathsf{pingpongT}.\ \lambda z : \mathsf{nat}.$
4. $\quad\quad\quad \mathsf{if}\ z\ \mathsf{is}\ \mathsf{Suc}\ z'\ \mathsf{then}$
5. $\quad\quad\quad\quad \mathbf{ret}\ [\mathbf{abort}\ k\ (\mathsf{pong}; (f\ z'))]$
6. $\quad\quad\quad \mathsf{else}\ \mathbf{ret}\ [\mathsf{pong}])\ n;$
7. $\quad \mathsf{ping};$
8. $\quad \mathbf{cmd}\ c).$

where

$\mathsf{pingpongT} \triangleq \Pi z : \mathsf{nat}.$
$[v\ w\ p\ h].\ \mathsf{SK}^*\ \{x \mapsto v+p\ * y \mapsto w+p * \mathsf{this}\ h$
$\curlywedge\ j \in x \mapsto v * \mathsf{this}\ h * y \mapsto w\ \curlywedge\ z \leq n$
$\curlywedge\ p+z = n\}\ \Sigma\bullet\mathsf{SK}\ (\,)$
$\{r.\ x \mapsto v+p\ * y \mapsto v+p * \mathsf{this}\ h$
$\curlywedge\ \mathsf{spec}\ r \sqsubseteq (x \mapsto v+p+1 * y \mapsto v+p$
$* \mathsf{this}\ h,$
$\mathsf{if}\ p = n\ \mathsf{then}$
$x \mapsto v+n+1 *$
$y \mapsto w+n+1 * \mathsf{this}\ h$
$\mathsf{else}\ \bot)^*\}$

Figure 8. ping–pong cooperation.

This behaviour is reflected in the type of ping–pong: when the function is executed in a heap containing at least the pointers x and y storing some natural number, the result after $n+1$ executions of ping and pong is a heap with the same shape, where the values of x and y are both incremented $n+1$ times.

The type invariant pingpongT describes the specification of the loop that defines the closure f described above: on each iteration of the recursive call, we insert calls to pong deeper into the closure, which will execute after the corresponding ping. Then, we make explicit that on each recursive call, the values stored in the heap have been incremented appropriately. We introduce a (box) logical variable p to account for this fact: when the recursive call to f occurs, p calls to ping and p calls to pong have occurred. The recursive call produces a closure whose spec we define using $\sqsubseteq$. The closure should be run in a heap after the $(p+1)$-execution of ping. As for the postcondition, If $p \neq n$, then the closure's head is a call to **abort**, line 5, and the postcondition is $\bot$. If $p = n$, i.e. this is the last iteration of f, then the closure corresponds to the one in line 6, entailing that this is the last execution of the loop, if $n > 0$

or that there was no loop at all otherwise. Hence, the final heap of the closure results $x \mapsto v + n + 1 * y \mapsto w + n + 1 * \mathsf{this}\, h$.

Unlike the case of **inc3**, the continuation object k here is aborted to more than once. As a result, the precondition in Kont^* has to accommodate for the changes in the state in each of the different jumping points. This is solved by using a (diamond) logical variable p, which allows us to discriminate the changes of each particular jumping point wrt. the captured heap j. The precondition in k states that the calls to **abort** occur when $n > 0$. Then, the heap at each jumping point should reflect the effect of $p + 1$ executions of ping and p executions of **pong**. The postcondition states that the finalisation code performs $p + 1$ execution of pong, and that it returns a closure which, again, is meant to run after the next ping. The postcondition in the closure is similar to the one in pingpongT, albeit the current iteration being $p + 1$ rather than p, as one ping has already executed.

In an online Appendix [12], we present a detailed proof outline for ping–pong. The Coq script can be found in the source files.

7. Discussion and related work

***Reasoning with non-algebraic* callcc** To understand the difference in specification and reasoning between algebraic and non-algebraic control operators, we have repeated our formal development for a non-algebraic set of operators, which we also make available online [12]. As a basis, we gave control operators a more familiar type for *non-parameterised* continuation monads [49]:

$$\begin{array}{l}\mathsf{callcc} : ((A \to \mathsf{SK}\, B) \to \mathsf{SK}\, A) \to \mathsf{SK}\, A \\ \mathsf{throw} : (A \to \mathsf{SK}\, B) \to A \to \mathsf{SK}\, B\end{array}$$

Continuation objects are ordinary side-effectful functions of type $A \to \mathsf{SK}\, B$, which do not make provisions for finalisation code, and are thus not algebraic [22]. The same remark applies to the type given to the $\mathcal{C}$-operator in [48], which is a different, but closely related control operator [15, 14].

We managed to soundly parameterise the monad with Hoare-style assertions using the following typing rules.

$$\frac{j,\ f : \Pi\, x{:}A.\ \mathsf{SK}\ B\ (S\, x\, j,\ \lambda\, r\, i\, m.\ \mathsf{False}) \vdash e : \mathsf{SK}\ A\ (P\, j, Q\, j) \qquad \forall j.\ (P\, j, Q\, j) \sqsubseteq (\lambda\, i.j = i \wedge R\, i, S)}{\vdash\ \mathbf{callcc}_j\ f.\, e : \mathsf{SK}\ A\ (R,\ S)}$$

$$\frac{\vdash f : \Pi\, x{:}A.\ \mathsf{SK}\ B\ (R,\ \lambda\, r\, i\, m.\ \mathsf{False}) \qquad \vdash e : A}{\vdash\ \mathbf{throw}\ f\, e : \mathsf{SK}\ B\ (R,\ \lambda\, r\, i\, m.\, \mathsf{False})}$$

The intuition for the **callcc** rule is that the user must provide the ending specification pair (R, S). The (binary) postcondition S for the whole command is used as a precondition for the continuation object f, after S is first instantiated with the value x that is passed to f, and the captured heap j. The rule has a side condition requiring that the specification $(P\, j, Q\, j)$ inferred for the body e, can be weakened into the desired (R, S), under the knowledge that the captured heap j equals the initial heap i.

The requirement that the specification (R, S) has to be provided by hand, practically differentiates the algebraic and non-algebraic operators. In the non-algebraic **callcc**, the specification is monolithic, and the postcondition S is usually a disjunction whose cases specify both the jumping and the non-jumping behaviour of the code. The algebraic **callcc** separates the two cases; the jumping is manually specified in the type of f, but the non-jumping specification can often be inferred by the typing rules from the structure of e, say, if e is straight line code, or by using the invariants provided with the loops in e, otherwise. Our examples rember-up-to-last and ping–pong illustrate the point, whereby e's specification directly corresponds to the supplied loop invariants. In the non-algebraic case, specifying these two examples incurs an overhead that the same annotation has to be provided twice; once as the loop invariant, and again as part of the specification of **callcc**.

Small vs. large footprints The need for large footprints and explicit naming of residual heaps arises in $\mathsf{HTT}_{\mathbf{cc}}$ due to the control operators. The $\mathsf{HTT}_{\mathbf{cc}}$ typing rule for ABORT, requires first discharging a precondition that the heap i at the point of the jump is related by the initial condition R to the heap j at the point of continuation capture. R is an ordinary predicate on heaps, rather than a Hoare triple. Thus, the usual idea of Separation logic, whereby a Hoare triple leaves the unused parts of a program implicitly unchanged, does not directly apply, and we have to name the unused parts in i and j in order to explicitly state their equality.

We have considered an alternative whereby the denotational semantics of control operators may automatically determine the unused part of the heap, by subtracting out of the current heap the portion described by the assertions. However, for this to work, the assertions would have to be *precise*, i.e., uniquely determine the portion to be subtracted. But then, the precision of each used assertion has to be formally proved. Thus, we opted for slightly increasing the specification burden by introducing explicit $\mathsf{this}\, h$ predicates, as a trade-off for not having to prove precision of assertions.

Despite the slight overhead of large footprints, we have not found them too problematic in practice. The naming of the residual heaps is intuitive and can be done systematically. Moreover, the logical variables used in the naming are *local* to the Hoare triple. This makes a big difference from ordinary first-order Hoare or Separation logic, where the global nature of logical variables makes such a naming scheme—and correspondingly, the use of large footprints—a complete non-starter. But mostly, it is the presence of separating conjunction $*$, which makes $\mathsf{HTT}_{\mathbf{cc}}$ capable of reasoning about heap disjointness with the same ease inherent in Separation logic, irrespective of whether the annotations describe full or partial heaps.

Specification-only variables and implicit constructions Our **callcc** primitive is indexed by a specification-only variable j, which binds the heap at the point of continuation capture. j should be used only in the assertions and proofs, but not in the executable parts of the **callcc** block. Unfortunately, Coq (and consequently $\mathsf{HTT}_{\mathbf{cc}}$) does not currently provide any means for enforcing this syntactic distinction. Declaring variables such as j as specification-only *is* natively supported by Coq*, an extension of Coq based on the *Implicit Calculus of Constructions* [3]. In the future, we plan to explore embedding $\mathsf{HTT}_{\mathbf{cc}}$ into Coq*, to make use of this feature.

***Higher-order heaps and semantic models for* callcc** Dreyer et al. [13] and Støvring and Lassen [41] develop semantics models and methods for equational reasoning in such models, for programs with continuations and mutable store. A specific focus in both works is on higher-order heaps [26, 50, 39]; that is, the ability to store computations (and continuations as a special case) into the heap. $\mathsf{HTT}_{\mathbf{cc}}$'s model is much simpler in this particular respect. While it allows programs that return continuations, and closures encapsulating continuations, it does not allow programs that store side-effectful computations into the heap. The reason is that we defined SK and Kont types *in terms of* heap, rather than *mutually recursively with* heap, as required for stored computations. In the future, we will develop a model for $\mathsf{HTT}_{\mathbf{cc}}$ with stored computations. We plan to build on the model for HTT by Svendsen et al. [43], which includes higher-order heaps, but no control operators. Our ultimate aim is to implement proper coroutines, following Reppy [35, cp. 10], and use them to verify concurrency primitives. Moreover, our inference of weakest pre- and strongest postconditions by the rules presented in Figure 4 is also in the spirit of *characteristic formulae* [32, 1, 8].

Hoare logics for higher-order control Crolard and Polonowski [10] have recently developed a Hoare logic for control operators, in which specifications are carried out in types. While in this respect, the approach is similar to $\mathsf{HTT}_{\mathbf{cc}}$ from the high-level point of view, there is a number of differences. For example, Crolard and Polonowski only consider mutable stack variables with block scope, but no pointers or aliasing. Procedures are not allowed to contain free variables, and type dependencies contain first-order data only, such as natural numbers. In contrast, in $\mathsf{HTT}_{\mathbf{cc}}$, we allow the full expressiveness of CIC, including Σ-types over specifications, which, as we illustrated, is required for specifying closures that return captured continuations. Berger [5] presents a first-order Hoare logic for **callcc** in an otherwise purely functional language. One of the main features of the logic is the polarity distinction between the types of programs that perform jumps ('jumping-to') and the types of labels for jumps ('being-jumped-to'). From the point of view of reasoning, the logic allows nesting Hoare triples inside the assertions. This is necessary for specifying closures with captured continuations, and achieves the same effect as Σ-types over specifications in our dependently-typed setting.

Hoare reasoning through dependent types Related systems that employ Hoare-style specification via types are HTT [30] and F^* [44]. HTT is a direct precursor of $\mathsf{HTT}_{\mathbf{cc}}$, but does not include the control operators. It uses an embedding of (small footprint) Separation logic via monads into Coq to formulate annotations and discharge verification conditions. A similar idea of Hoare monads in Coq, without control operators, has also been considered by Swierstra [45]. F^* specifies computations using a somewhat different monad from the above work. Instead of postconditions ranging over the input and output heaps, F^* considers predicate transformers ranging over *sets* of input and output heaps. F^* does not include a separate form of preconditions to specify safety; thus, its Hoare logic is not fault-avoiding as is $\mathsf{HTT}_{\mathbf{cc}}$, or other systems based on Separation logic. F^* relies on Z3 for automatic discharge of verification conditions. In order to facilitate automation, its assertion logic is a first-order fragment supported by Z3. F^* does not consider **callcc**, or other abstractions required by it, such as Σ-types over specifications.

CPS translation CPS translation in the case of dependent types has been studied by Barthe and Uustalu [4] who show the impossibility of CPS-translating dependent inductive and Σ-types. As $\mathsf{HTT}_{\mathbf{cc}}$ essentially relies on Σ-types to encode nested Hoare triples – as inhabitants of the $\Sigma{\bullet}\mathsf{SK}$ type – it seems impossible to present $\mathsf{HTT}_{\mathbf{cc}}$ as a CPS translation into a **callcc**-free fragment of $\mathsf{HTT}_{\mathbf{cc}}$.

8. Conclusions

In this paper, we have presented $\mathsf{HTT}_{\mathbf{cc}}$, a higher-order type theory for verification of programs with **callcc** control operators. $\mathsf{HTT}_{\mathbf{cc}}$ supports mutable state in the style of Separation logic, and, to the best of our knowledge, is the first Hoare logic or type theory to support the combination of higher-order functions, mutable state and control operators. The support for mutable state comes with a twist, however. While our assertion logic embeds separating conjunction $*$, we used large footprint specification style, which we found necessary to relate heaps captured with the continuation to heaps at the point of a jump. We use *algebraic* control operators, initially introduced by Jaskelioff [22], which we here adapt to non-rollbackable state. We argue that in practice, the algebraic operators require less manual program annotations, than the non-algebraic variants. We have implemented $\mathsf{HTT}_{\mathbf{cc}}$ as an embedding in Coq, and verified a number of characteristic example programs that use **callcc** [12].

Acknowledgements We thank the anonymous referees for their helpful feedback and comments on the paper. We also thank Mauro Jaskelioff for the fruitful discussions about algebraic effects.

This research was partially supported by the Spanish MINECO projects TIN2010-20639 Paran10 and TIN2012-39391-C04-01 Strongsoft, AMAROUT grant PCOFUND-GA-2008-229599, and Ramon y Cajal grant RYC-2010-0743.

References

[1] ACETO, L., AND INGÓLFSDÓTTIR, A. Characteristic formulae: From automata to logic. *Bulletin of the EATCS 91* (2007).

[2] ARBIB, M. A., AND ALAGIC, S. Proof rules for gotos. *Acta Inf. 11* (1979).

[3] BARRAS, B., AND BERNARDO, B. The implicit calculus of constructions as a programming language with dependent types. In *FoSSaCS* (2008).

[4] BARTHE, G., AND UUSTALU, T. CPS translating inductive and coinductive types. In *PEPM* (2002).

[5] BERGER, M. Program logics for sequential higher-order control. In *FSEN* (2009).

[6] BERTOT, Y., AND CASTÉRAN, P. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. 2004.

[7] BJØRNER, D., AND JONES, C. B., Eds. *The Vienna Development Method: The Meta-Language* (1978), vol. 61 of *LNCS*.

[8] CHARGUÉRAUD, A. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris-Diderot, 2010.

[9] CLINT, M., AND HOARE, C. A. R. Program proving: Jumps and functions. *Acta Inf. 1* (1972).

[10] CROLARD, T., AND POLONOWSKI, E. Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control. *J. Log. Algebr. Program. 81*, 3 (2012).

[11] DANVY, O., AND FILINSKI, A. Representing control: A study of the CPS transformation. *MSCS 2*, 4 (1992).

[12] DELBIANCO, G. A., AND NANEVSKI, A. Supporting material. http://software.imdea.org/~germand/HTTcc, March 2013.

[13] DREYER, D., NEIS, G., AND BIRKEDAL, L. The impact of higher-order state and control effects on local relational reasoning. In *ICFP* (2010).

[14] FELLEISEN, M., FRIEDMAN, D. P., DUBA, B., AND MERRILL, J. Beyond Continuations. Tech. Rep. 216, Indiana University, 1987.

[15] FELLEISEN, M., FRIEDMAN, D. P., KOHLBECKER, E. E., AND DUBA, B. F. Reasoning with continuations. In *LICS* (1986).

[16] FELLEISEN, M., WAND, M., FRIEDMAN, D., AND DUBA, B. Abstract continuations: a mathematical semantics for handling full jumps. In *LISP and functional programming* (1988).

[17] FILINSKI, A. Representing monads. In *POPL* (1994).

[18] FRIEDMAN, D. P., AND FELLEISEN, M. *The Seasoned Schemer*. 1996.

[19] GONTHIER, G., MAHBOUBI, A., AND TASSI, E. A Small Scale Reflection Extension for the Coq system. Tech. Rep. 6455, INRIA, 2008.

[20] GRIFFIN, T. A formulae-as-types notion of control. In *POPL* (1990).

[21] HYLAND, M., LEVY, P. B., PLOTKIN, G. D., AND POWER, J. Combining algebraic effects with continuations. *TCS 375*, 1-3 (2007).

[22] JASKELIOFF, M. Modular monad transformers. In *ESOP* (2009).

[23] JENSEN, J. B., BENTON, N., AND KENNEDY, A. High-level separation logic for low-level code. In *POPL* (2013).

[24] KLEYMANN, T. Hoare logic and auxiliary variables. *Formal Aspects of Computing 11* (1999).

[25] KOWALTOWSKI, T. Axiomatic approach to side effects and general jumps. *Acta Inf. 7* (1977).

[26] KRISHNASWAMI, N. R. *Verifying Higher-Order Imperative Programs with Higher-Order Separation Logic*. PhD thesis, Carnegie Mellon University, 2011.

[27] THE COQ DEVELOPMENT TEAM. *The Coq proof assistant reference manual*. TypiCal Project, 2012. Version 8.4.

[28] MOGGI, E. Computational lambda-calculus and monads. In *LICS* (1989).

[29] NANEVSKI, A., MORRISETT, J. G., AND BIRKEDAL, L. Hoare type theory, polymorphism and separation. *JFP 18*, 5-6 (2008).

[30] NANEVSKI, A., VAFEIADIS, V., AND BERDINE, J. Structuring the verification of heap-manipulating programs. In *POPL* (2010).

[31] O'HEARN, P. W., REYNOLDS, J. C., AND YANG, H. Local reasoning about programs that alter data structures. In *CSL* (2001).

[32] PARK, D. M. R. Concurrency and automata on infinite sequences. In *Theoretical Computer Science* (1981).

[33] PLOTKIN, G. D., AND POWER, A. J. Computational effects and operations: An overview. *ENTCS 73* (2004).

[34] PLOTKIN, G. D., AND POWER, J. Algebraic operations and generic effects. *Applied Categorical Structures 11*, 1 (2003).

[35] REPPY, J. H. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[36] REYNOLDS, J. C. The discoveries of continuations. *LISP and Symbolic Computation 6*, 3-4 (1993).

[37] REYNOLDS, J. C. Separation logic: A logic for shared mutable data structures. In *LICS* (2002).

[38] SAABAS, A., AND UUSTALU, T. A compositional natural semantics and Hoare logic for low-level languages. *TCS 373*, 3 (2007).

[39] SCHWINGHAMMER, J., BIRKEDAL, L., REUS, B., AND YANG, H. Nested Hoare triples and frame rules for higher-order store. *LMCS 7*, 3 (2011).

[40] SPRINGER, G., AND FRIEDMAN, D. P. *Scheme and the Art of Programming*. MIT Press and McGraw-Hill, 1989.

[41] STØVRING, K., AND LASSEN, S. B. A complete, co-inductive syntactic theory of sequential control and state. In *POPL* (2007).

[42] STRACHEY, C., AND WADSWORTH, C. P. Continuations: A mathematical semantics for handling full jumps. *HOSC 13*, 1/2 (2000).

[43] SVENDSEN, K., BIRKEDAL, L., AND NANEVSKI, A. Partiality, state and dependent types. In *TLCA* (2011).

[44] SWAMY, N., WEINBERGER, J., SCHLESINGER, C., CHEN, J., AND LIVSHITS, B. Verifying higher-order programs with the Dijkstra monad. In *PLDI* (2013).

[45] SWIERSTRA, W. A hoare logic for the state monad. TPHOLs '09.

[46] TAN, G., AND APPEL, A. W. A compositional logic for control flow. In *VMCAI* (2006).

[47] THIELECKE, H. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edimburgh, 1997.

[48] THIELECKE, H. Control effects as a modality. *JFP 19*, 1 (2009).

[49] WADLER, P. Monads and composable continuations. *LISP and Symbolic Computation 7*, 1 (1994).

[50] YOSHIDA, N., HONDA, K., AND BERGER, M. Logical reasoning for higher-order functions with local state. *LMCS 4*, 4 (2008).